

Real-time Operating Systems and Systems Programming

Threads

Definition of Thread

- A thread is a unit of execution, associated with a process, with its own thread ID, stack, stack pointer, program counter, condition codes, and general-purpose registers.
- Multiple threads associated with a process run concurrently in the context of that process, sharing its code, data, heap, shared libraries, signal handlers, and open files.

Process vs Thread

- Process – unit of resource ownership:
 - a virtual address space which holds the process image.
 - protected access to processors, other processes, files, and I/O resources.
- Thread – unit of dispatching:
 - Has an execution state (running, ready, etc.)
 - Saves thread context when not running
 - Has an execution stack and some per-thread static storage for local variables
 - Has access to the memory address space and resources of its process

Benefits of using threads instead of processes

- Properly implemented, threads take:
 - Less time to create a new thread than a process, because the newly created thread uses the current process address space.
 - Less time to terminate a thread than a process.
 - Less time to switch between two threads within the same process, partly because the newly created thread uses the current process address space.
 - Less communication overheads -- communicating between the threads of one process is simple the threads share almost everything: address space, in particular. So, data produced by one thread is immediately available to all the other threads.

Benefits of multi-threading

- Improve application responsiveness
- Use multiprocessors more efficiently
- Improve program structure
- Use fewer system resources

Thread Libraries

- Provide interface for thread manipulation:
 - creating and destroying threads
 - passing messages and data between threads
 - scheduling thread execution
 - saving and restoring thread contexts
- Are not a part of C standard
- Example libraries:
 - POSIX threads
 - SOLARIS threads

Thread Control

- Pthreads defines about 60 functions that allow C programs to create, kill, and reap threads, to share data safely with peer threads, and to notify peers about changes in the system state.
- However, most threaded programs use only a small subset of the functions defined in the interface.

Threaded Hello.c

```
#include <pthread.h>
#include <stdio.h>
void *thread(void *vargp);

int main() {
    pthread_t tid;

    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, NULL);
    exit(0);
}

/* thread routine */
void *thread(void *vargp) {
    printf("Hello, world!\n");
    return NULL;
}
```


Creating threads

```
#include <pthread.h>
```

```
typedef void *(func)(void *);
```

```
int pthread_create(pthread_t *tid,  
pthread_attr_t *attr, func *f,  
void *arg);
```

returns: 0 if OK, non-zero on error

```
pthread_t pthread_self(void);
```

Terminating Threads

A thread terminates in one of the following ways:

- The thread terminates *implicitly* when its top-level thread routine returns.
- The thread terminates *explicitly* by calling the `pthread_exit()` function, which returns a pointer to the return value thread return. If the main thread calls `pthread_exit`, it waits for all other peer threads to terminate, and then terminates the main thread and the entire process with a return value of thread return.
- Some peer thread calls the Unix `exit()` function, which terminates the process and all threads associate with the process.
- Another peer thread terminates the current thread by calling the `pthread_cancel()` function with the ID of the current thread.

`int pthread_exit(void *thread_return);`

- Returns 0 if OK, nonzero on error

`int pthread_cancel(pthread_t tid);`

- Returns 0 if OK, nonzero on error

Reaping terminated threads

- Threads wait for other threads to terminate by calling the `pthread_join` function.
- `int pthread_join(pthread_t tid, void **thread_return);`
- The `pthread_join` function blocks until thread `tid` terminates,
- There is no way to instruct `pthread_join` to wait for an arbitrary thread to terminate.

Detaching threads

- At any point in time, a thread is *joinable* or *detached*. A joinable thread can be reaped and killed by other threads. Its memory resources (such as the stack) are not freed until it is reaped by another thread.
- In contrast, a detached thread cannot be reaped or killed by other threads. Its memory resources are freed automatically by the system when it terminates.
- By default, threads are created joinable. In order to avoid memory leaks, each joinable thread should either be explicitly reaped by another thread, or detached by a call to the `pthread_detach` function.
- `int pthread_detach(pthread_t tid);`
- Note:
`pthread_detach(pthread_self()) // used to detach self`
- Generally threads are detached

Shared variables

- Sharing variables is one of the most attractive features of threads
- It is also most dangerous for creating bugs that are difficult to detect
- Global variables are shared
- Local automatic variables (stack) are not shared but are not protected either (share common virtual address space)
- Local static variables are shared as globals
- Generally: a variable is shared if and only if one of its instances is referenced by more than one thread.

Incorrect sharing

```
#include <pthread.h>
#define NITERS 100000000
void *count(void *arg);
/* shared variable */
unsigned int cnt = 0;
int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, count, NULL);
    pthread_create(&tid2, NULL, count, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
}
void *count(void *arg) { // thread routine
    int i;
    for (i=0; i<NITERS; i++)
        cnt++;
    return NULL; }
```

Sharing problem

Code for thread:

```
for (i=0; i<NITERS; i++)  
    ctr++;
```

Is actually:

```
LOAD  ctr  
INCREMENT  ctr  
STORE  ctr
```

Mutexes

- A mutex is synchronization variable that is used to protect the access to shared variables. There are three basic operations defined on a mutex.
 - Init, Lock, Unlock
- `int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);`
- Compile time initialization
`pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER;`

Mutex lock and unlock

- `int pthread_mutex_lock(pthread_mutex_t *mutex);`
- `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
- These are atomic operations
- Locking is also called acquiring the mutex, unlocking is called releasing
- At any moment only one thread can hold a mutex

Using mutexes

```
// general code  
pthread_mutex_t mutex;  
pthread_mutex_init(&mutex, NULL);
```

```
// thread code  
pthread_mutex_lock(&mutex);
```

```
// critical section  
// access shared variables  
pthread_mutex_unlock(&mutex);
```

Correct thread routine

```
/* thread routine */  
void *count(void *arg)  
{  
    int i;  
  
    for (i=0; i<NITERS; i++) {  
        pthread_mutex_lock(&mutex);  
        cnt++;  
        pthread_mutex_unlock(&mutex);  
    }  
    return NULL;  
}
```

Deadlocks

- Locking order might cause issues when threads hold mutexes mutually